

Debreceni Egyetem

Informatikai Kar

Biológiai géninterakciók feldolgozása
Apache Hadoop segítségével

Témavezető:

Dr. Antal Bálint

egyetemi adjunktus

Készítette:

Auer Bence

mérnök informatikus

Debrecen

2014

Tartalomjegyzék

1. Bevezetés.....	3
2. Az Apache Hadoop	5
2.1 Architektúra - HDFS.....	5
2.2 A MapReduce paradigma	8
2.2.1 A Map és a Reduce	8
2.2.2 Működése.....	8
2.2.3 Életciklusa	10
2.3 Kiegészítő lehetőségek	10
3. BioGrid Hadoop.....	12
3.1 Felhasznált eszközök	12
3.1.1 Operációs Rendszer.....	12
3.1.3 Hadoop Streaming.....	15
3.1.4 A Python és az EasyGUI	16
3.2 A Repository bemutatása	17
3.2.1 GUI (start.py).....	19
3.2.2 Mapper (mapper.py).....	25
3.2.3 Reducer (reducer.py).....	28
3.2.4 custom.jar (CustomCSVFormat.java)	28
3.3 Tesztelés.....	31
4. Összefoglalás	35
5. Irodalomjegyzék	36
6. Ábrajegyzék.....	37
7. Eredmények	39

1. Bevezetés

A napjainkban folyó tudományos kutatások és felfedezések elképzelhetetlennek tűnnének valamilyen szintű informatikai támogatás nélkül. Gondoljunk bele, hogy egy részecskegyorsítóban vagy egy genetikai laborban milyen óriási mennyiségű adatot kell kezelni és elemezni ahhoz, hogy csak a számunkra fontos eredményeket láthassuk. A *CERN Nagy Hadronütköztetőjében* például egy évben hozzávetőlegesen 25 petabájtnyi adat generálódik, ennek feldolgozásához pedig a hagyományos adatfeldolgozási technológiák már sajnos nem elegendők. Az ilyen és ehhez hasonló méretű nagy adathalmazokat az informatika "Big Data"-nak hívja és az ezen alapuló tudományt "Big Data Science"-ként emlegeti. Gyökerei csupán néhány évre nyúlnak vissza, de például Barack Obama amerikai elnök 2012-es újraválasztását is már sokan a Big Data-hoz kötik, ugyanis a választás évében tekintélyes összeggel támogatta a Big Data irányú R&D fejlesztéseket a kampányának segítése érdekében. Természetesen több informatikai nagyvállalat is él a Big Data által nyújtott lehetőségekkel, többek között a *Facebook*, az *eBay* és az *Amazon.com* is. Ők többnyire a felhasználói igények felmérésére, marketing célokra és piackutatásra használják ezt a technológiát.

Tanulmányaim során mindig is az adatbázisrendszerek és azok technológiai foglalkoztattak leginkább, ezért is esett a témaválasztásom a nagy adathalmazok feldolgozására. A terület újdonsága és a felfedezetlen lehetőségek mind-mind motiváltak ahhoz, hogy közelebbről is megismerkedjek a Big Data Science-el.

Mindenekelőtt egy olyan rendszerre volt szükségem, amely hatékonyan képes nagy adattömegeket kezelni. Választásom az *Apache Hadoop* keretrendszerre esett, amely lehetővé tette az adatok elosztott fájlrendszeren (*klaszter* környezetben) történő tárolását és feldolgozását, emellett kiváló lehetőséget nyújtott az egyszerű programozási modellen (*MapReduce*) keresztüli Big Data alkalmazásfejlesztésnek is.

Elsődleges célom egy olyan alkalmazás megírása volt, amely biológiai géninterakciók lekérdezését és rendezését valósítja meg, mindehhez felhasználva a Hadoop és a Big Data által nyújtott lehetőségeket. Másodlagos célom pedig az *Apache Hadoop* architektúrájának a bemutatása volt, mivel jelenleg magyar nyelven ilyen írás eddig sajnos nem létezett. A dolgozat logikai szerkezete miatt először a Hadoop keretrendszert és annak fontosabb komponenseit ismertetem, majd ezt követi az alkalmazásom dokumentációja, amely

remélhetőleg a biológiai kutatások egyik hasznos eszköze lesz. A szoftver a BioGRID nevű adatbázis egyik webszolgáltatását használja, amellyel fehérje-fehérje és genetikai interakciókra vonatkozó információkat tudunk lekérdezni. A fejlesztést egy virtuális gépen végeztem, amelyre külön telepítettem a Hadoop-infrastruktúrát. A rendszer skálázhatósága már itt megmutatkozott, hiszen *single-node*-os klaszterként (azaz "egygépes" klaszterként) tudtam a teljes rendszert használni. Az alkalmazás nagy része Python nyelven íródott, viszont ezen kívül tartalmaz egy kevés Java kódot is. A könnyű használhatóság jegyében egy egyszerű grafikai felület (GUI) is implementálva lett a projekthez.

2. Az Apache Hadoop

Az Apache Hadoop egy nyílt forráskódú keretrendszer, amely elosztott rendszereken futó adat-intenzív alkalmazásokat támogat. Alapjait még 2004-ben kezdte el fejleszteni *Doug Cutting* és *Mike Cafarella*. Eredeti céljuk egy általuk már megírt webes keresőmotornak (a *Nutch*-nak) az elosztott rendszerre való implementálása volt, amelyhez a *Google* által is használt *Google File System*-et, azaz a *GFS*-t vették alapul. Az így született fájlrendszert *Nutch Distributed Filesystem*-nek (*NDFS*-nek) nevezték el. A fejlesztés itt még nem ért véget. Nem sokkal azután, hogy a *Google* publikálta a *GFS* architektúráját, megismertette a világgal a *MapReduce* paradigmát is, amely arra ösztönözte a *Nutch* csapatot, hogy a saját motorjukba is beépítsék azt. A készítőik úgy gondolták, hogy az így megszületett rendszer nem csak a kereső esetében, hanem más területeken is jól alkalmazhatóvá válhat, így 2006 februárjában egy külön projektet indítottak *Hadoop* néven, hogy a rendszer eleget tudjon tenni napjaink (leginkább webes) Big Data követelményeinek. Ebben az időben *Cutting* a *Yahoo!*-nál dolgozott, ami biztosítani tudta számára a megfelelő erőforrásokat és fejlesztőcsapatot, hogy a *Hadoop* egy valóban kifinomult rendszer lehessen. 2008 februárjában a *Yahoo!* bejelentette, hogy a keresőmotorjuk indexeit egy 10,000 magos *Hadoop*-cluster generálja. Innentől fogva az alkalmazás kiemelt *Apache* projekt lett és a *Yahoo!* után számos vállalat kezdte el a saját *Hadoop*-cluster-ét kiépíteni, többek között a *New York Times* és a *Last.fm* is.

A *Hadoop* valódi előnye a *teljesítményében*, *megbízhatóságában*, *skálázhatóságában* és *könnyű menedzselhetőségében* rejlik. Ahhoz, hogy ezeket az előnyöket mi is ki tudjuk használni, először mindenképp érdemes közelebbről megismerkednünk a rendszer architektúrájával. (Az alábbi *HDFS*-ről és a *MapReduce*-ről szóló fejezetek nagyrészt a *Yahoo!* által kidolgozott *Hadoop Tutorial* alapján készültek.)

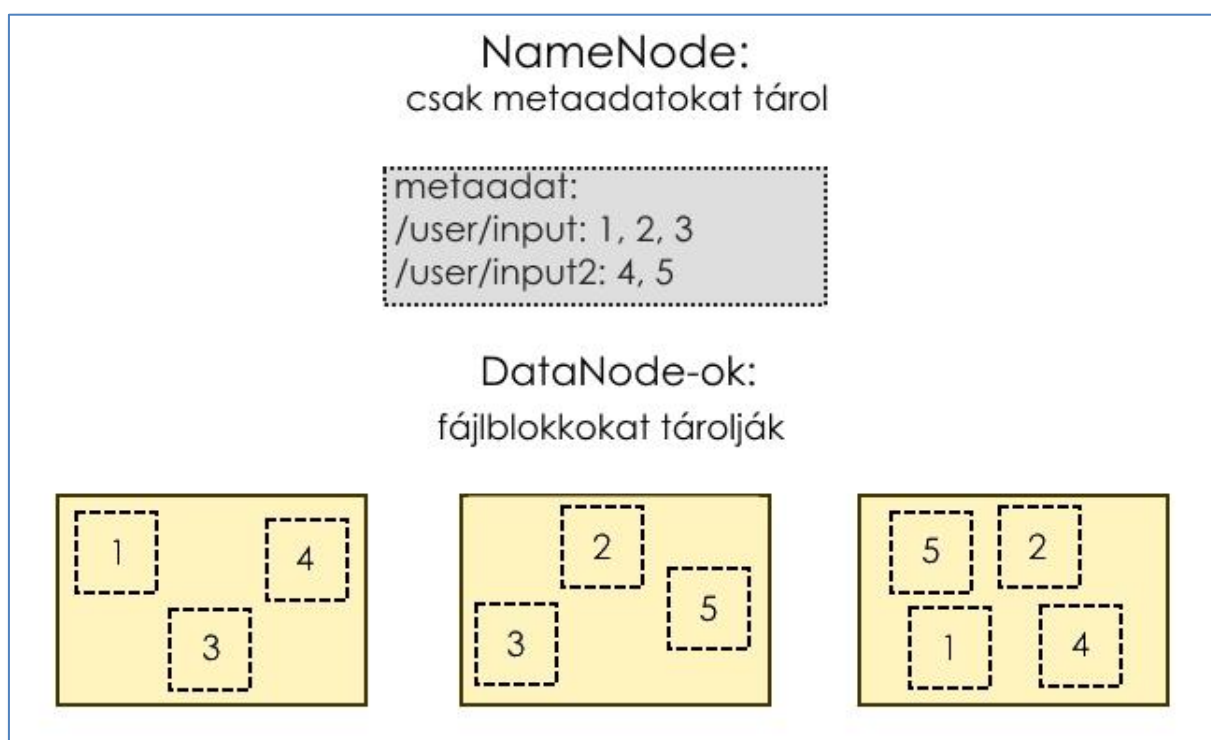
2.1 Architektúra - HDFS

A nagyméretű adathalmazokkal való munka meglehetősen bonyolult feladat. Egy efféle tevékenység adatmennyiségének alsó határa legalább néhány száz gigabájt nál kezdődik és akár több terra- és petabájtnyi méreteket ölthet. Szerencsére a *Hadoop* az ekkora adatmennyiségeket kiválóan kezeli, viszont az ilyen méretű adatokon való számításokhoz

sajnos egy hétköznapi számítógép erőforrása nem elegendő, csakis valamilyen grid-infrastruktúra jöhet szóba.

A Hadoop ennek megoldására egy elosztott fájlrendszert, a *HDFS-t* (*Hadoop Distributed File System-et*) implementálja, melynek egyik feladata az, hogy a nagyméretű bemeneti fájlokat feldarabolja és a darabokat egy klaszter-rendszer egyes csomópontjaihoz (*node*-jaihoz) eljuttassa. Ekkor a rendszer *párhuzamosan* dolgozhat az adatainkon, amivel jelentősen csökkenti a feldolgozásnak az idejét. Ehhez hasonló elosztott megoldások már a Hadoop előtt is léteztek (*NFS*), viszont egyszerű programozási modell (amellyel gyorsan lehetne fejleszteni és tesztelni) és automatikus feldarabolása az adatoknak eddig nem volt megoldott.

A rendszer redundanciát is biztosít, hiszen egy adatot alapesetben több *node*-on tárol el. (Például egy 100 *node*-os klaszter esetében, ha az egyik számítógép meghibásodik, annak feladatait egy másik *node* veszi át, mellyel az esetleges adatvesztéseket tudjuk elkerülni.)



1. ábra - A redundancia biztosításának érdekében a DataNode-ok egy fájlblokkot akár több helyen is el tudnak tárolni. Az ábrán látható esetben egy blokk két különböző *node*-on van jelen, tehát ez esetben a replikációs faktor 2.

Rekord-orientált rendszer lévén, a Hadoop a bemenő adatokat sorokra bontja le (természetesen az alkalmazás logikájának megfelelően meghatározhatjuk másképp is) és azokat az egyes csomópontokra juttatja el. A beolvasás így a helyi lemezről történik, ami

nagyban *csökkenti a hálózati terhelést*. Továbbá az, hogy egy adott számítási feladatot az adatokhoz (a node-okra) továbbítja és nem az adatokat a számításokhoz, nagyban növeli a rendszer *hatékonyságát és teljesítményét* is.

A keretrendszer nem képes bármilyen általunk készített programot elosztottan futtatni, ugyanis a kódnak szükséges követnie a MapReduce programozási modellt. A HDFS szempontjából ez csupán annyit jelent, hogy a kommunikációknak a száma előre meg van határozva az egyes processzek számára, ami a rendszer *megbízhatóságát* fokozza. Emellett másik fontos előnye a HDFS-nek a *skálázhatóság*, hiszen egyszerűen lehet hozzáadni és elvenni csomópontokat a klaszterhez a Hadoop teljes átkonfigurálása nélkül.

A HDFS egy blokk strukturált fájlrendszer: minden egyes fájl fix méretű blokkokból áll, melyek a klaszteren vannak eltárolva (egy vagy több node-on). Ezeket a csomópontokat *DataNode*-oknak (*DataNodes*-nak) nevezzük. Egy fájlhoz tartozó blokkok nem feltétlenül vannak egyazon háttértáron eltárolva, ezért a rendszernek külön koordinálni kell tudnia azt, hogy melyik *DataNode*-ról melyik blokkot fogja használni az adatok feldolgozásakor. Alapesetben a rendszer a blokkokat (a redundancia miatt) 3 csomóponton tárolja el, azonban ez az érték egy konfigurációs fájlban keresztül tetszőlegesen módosítható. A blokkok alaptermélete 64 MB, ami sokkal nagyobb a más fájlrendszereknél megszokott 4 vagy 8 kB-hoz képest. Ebből fakadóan kevés metaadatot kell tároljunk a fájlokról. Továbbá ezek a komponensek nem részei a hagyományos értelemben vett fájlrendszernek. Például egy *ls -lrth* parancsnak a kimenete csak a lokális fájlrendszeren található fájlakat fogja kilistázni, mivel a HDFS service-ek külön namespace-ben futnak. Ha az elosztott rendszeren lévő fájlakat szeretnénk elérni, ahhoz a Hadoop saját "*shell*"-jét szükséges használnunk, melynek parancsai nagyon hasonlóak az alapvető *Unix*-os parancsokhoz. A metaadatok tárolására elkülönített node-ot *NameNode*-nak nevezzük, amely lényegében az összes HDFS fájlra vonatkozó információt tartalmazza (fájlnév, elérési útvonal, jogosultságok, stb.). Kis mérete révén ezeket az adatokat a *NameNode* a saját memóriájában tárolja el, hogy azokat minél gyorsabban ellehessen érni. Egyik hátránya sajnos a Hadoop-nak, hogy a *NameNode* meghibásodása esetén egyetlen blokkját se fogjuk tudni elérni a fájlrendszernek, viszont ezek a node-ok alapvetően nincsenek kitéve nagy terhelésnek egy *DataNode*-hoz képest. (Nagy infrastruktúránál persze érthető, hogy a magas rendelkezésre állás miatt a *NameNode*-ot duplikálják.) *DataNode* meghibásodása esetén viszont a fent említett tárolási eljárások kiváló megoldást nyújtanak az adatok védelmére.

2.2 A MapReduce paradigma

A *MapReduce* programozási modellt nagy adathalmazok párhuzamos, elosztott rendszereken történő feldolgozására tervezték. Alapját a funkcionális programozás ihlette, annak is a *map* és *reduce* funkciói.

2.2.1 A Map és a Reduce

A funkcionális programozásban a *map* egy olyan magas szintű függvényt jelöl, amely egy lista minden egyes elemére alkalmaz egy függvényt, majd eredményként a már transzformált új listával tér vissza. Gyakran nevezik "*apply-to-all*" ("alkalmazd mindegyikre") függvénynek is. Például ha van egy listánk $[2, 5, 3, 4, 6, 7, 21]$, amelyre definiáltunk egy *negyzet* $x = x * x$ függvényt, akkor a *map negyzet* $[2, 5, 3, 4, 6, 7, 21]$ eredménye a $[4, 25, 9, 16, 36, 49, 441]$ lista lesz.

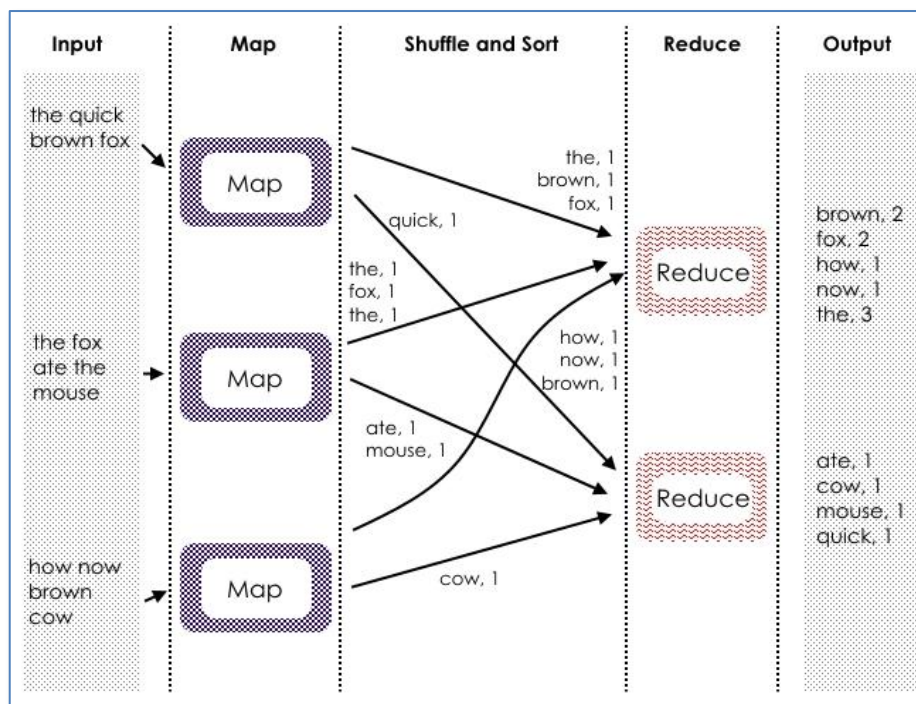
A *reduce* (vagy ismertebb nevén *fold*) egy szintén magas szintű függvény, amely egy szekvenciát redukál úgy, hogy annak mindegyik elemére rekurzívan alkalmaz egy függvényt. Legyen adott a következő függvényünk, *osszead* $= x + y$. Ebben az esetben (a *map*-nál szereplő példát alapul véve) a *reduce (osszead, [2, 5, 3, 4, 6, 7, 21])* eredménye 48 lesz.

A fentebb említett két függvényt külön-külön több programozási nyelv (például a Python) is implementálja. A *map* és *reduce* együttesen egy hatékony eszköz a strukturálatlan adatok feldolgozására.

2.2.2 Működése

A Hadoop esetében a *map* fázis (*mapper*) az adatok szűrését és párhuzamos futását valósítja meg – amelyhez szükséges bemeneti adatokat természetesen a HDFS szolgáltatja –, a *reduce* fázis (*reducer*) pedig a *map*-tól kapott darabolt adatokat transzformálja át végső formájukba, visszatérve a számunkra fontos kimenettel. A Hadoop ezen kívül tartalmaz más komponenseket is, amelyek a MapReduce (MR) implementálásához (és a hatékony használatához) szükségesek.

A bemeneti adatoknak a feldarabolását az *Input reader* végzi, amely darabokat a keretrendszer egy-egy mapper-hez rendel hozzá. Az MR általában kulcs-érték párokon szokta a számításokat végezni, így egy MR *job* (futás) bemeneti kulcs-érték párok listájából, kimeneti kulcs-érték párok listáját fogja létrehozni. A mapper és a reducer között szintén kulcs-érték párok generálódnak, amelyek az egyes reducer task-ok input-ját alkotják. Ezen kulcsoknak nem szükséges egyedieknek lenniük. Például egy "word count" programban - ahol egy szöveg összes szavának az előfordulását számláljuk -, a mapper a szöveg sorait szavakra bontja fel és kimenetként olyan kulcs-érték párokkal tér vissza, ahol a kulcs a szót, az érték pedig annak előfordulását jelöli (a szöveg egy sorában). Ezt követően a *Partition* (felosztó, particionáló) rész következik, ahol az egyes mapper kimeneteket egy meghatározott reducer-hez rendeljük hozzá. Ezt a folyamatot *Shuffling*-nak, azaz "keverésnek" nevezzük. Minden érték ahol a kulcsok megegyeznek, ugyanazon reducer-hez fog kerülni függetlenül attól, hogy valójában melyik mapper-től is származik. A reducer - a példánk esetében - az ugyanazon kulccsal rendelkező rekordokat (szavakat) összegezi és az alapján generálja ki az output fájlnkat, amelyet az *Output writer* fog a saját HDFS-ünkre írni, ami a szavak előfordulásának számait fogja tartalmazni. A példa MR job teljes működését a 2. ábra szemlélteti:

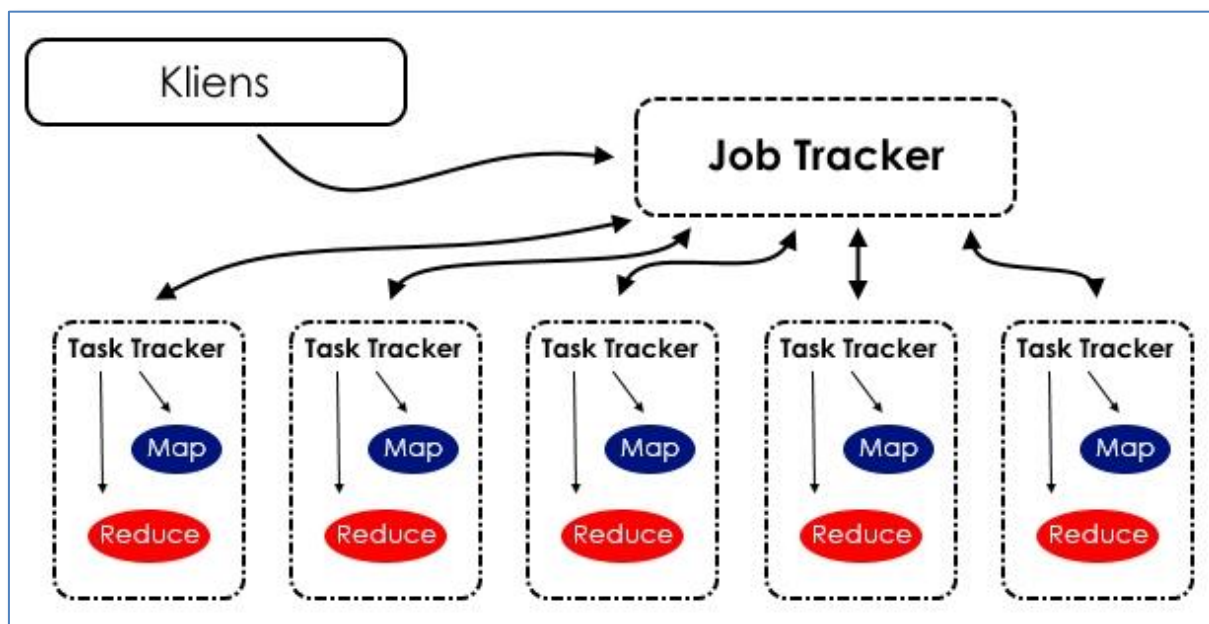


2. ábra - A "word count" program működése

Az input fájlnk feldarabolása után a mapper-ek kulcs-érték párokat hoznak létre, melyek a reducer-ek inputjait fogják alkotni. A Reduce funkció az azonos kulcsú rekordok összegzését fogja elvégezni, ami kimenetként a szavak előfordulásának számát fogja számunkra jelenteni.

2.2.3 Életciklusa

Amikor egy MR job-ot futtatunk, nagyon komplex folyamatok mennek végbe. A dolgozatnak nem célja a teljes MR életciklust bemutatni, csupán a megírt alkalmazás szempontjából fontos komponenseket szeretné kiemelni. Az első legfontosabb a *Job Tracker*, amely az egész program futtatásáért felelős. Daemon processzként az ő feladata szétosztani az egyes mapperek között az adatokat, hogy azok egy időben legyenek feldolgozva, valamint ezekhez a mapper-ekhez külön-külön *Task Tracker* processzeket rendel hozzá, hogy a mapper-ek rajtuk keresztül kapják meg a feldarabolt adatokat. Hasonló elven a reducer-ekkel is kommunikál, és az állapotukat folyamatosan riportálja a Job Tracker felé. Ezeken felül működik a *Job Client*, amely a bemenő adatok feldarabolásáért, az erőforrások elérhetőségéért és minden egyes új MR job validálásáért felelős.



3. ábra - A Job tracker és a Task Tracker kapcsolata

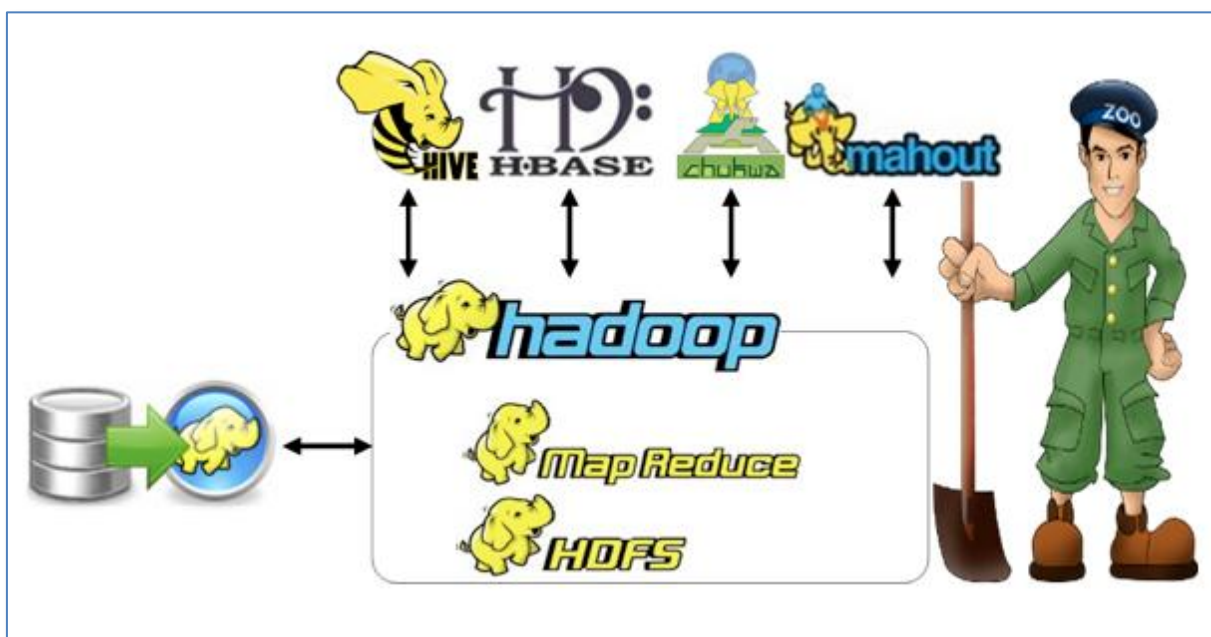
A Job Tracker daemon processzként működve a feldarabolt adatok szétosztásáért felelős, valamint a Task Tracker processzeket működteti, amely az egyes Map, Reduce és Shuffle folyamatok működéséhez szükséges adatokat biztosítja.

2.3 Kiegészítő lehetőségek

A Java alapú MapReduce programok a HDFS-el együtt egy nagyon hatékony keretrendszert alkotnak, de az azon keresztül programozás igen komplex ismereteket igényel.

Sok olyan fejlesztő van - például *Business Analyst*-ok, *Data Scientist*-ek - akik számára hasznos eszköz lehetne a Hadoop, ha nem kellene hozzá megfelelő szintű Java tudás. Az ilyen fejlesztőket nem érdekeli, hogy hány mapper és reducer van egy adott programhoz, inkább csak a számukra fontos munkára próbálnak fókuszálni. Ezt az igényt több MR implementáció segíti, például a *Hive* is.

A *Hive* egy *SQL-szerű* MapReduce implementáció, amely az "adattárházak" *SQL*-fejlesztők munkáját hivatott segíteni. Az *SQL*-t viszont strukturált adatok lekérdezésére szoktuk használni leginkább, így igény merült fel egy olyan eszköz kifejlesztésére, amely több flexibilitást nyújt az elosztott rendszeren tárolt adatok feldolgozását illetően. Erre egyik megoldása Yahoo! Research-nek a *Pig* nevű szkriptnyelv volt, amely megvalósította a deklaratív *SQL-szerű* lekérdezőnyelv és az alacsony szintű-, procedurális nyelv igényét, amellyel MapReduce kódot tudunk létrehozni. A sokféle végfelhasználó kiszolgálására további Hadoop projektek is létrejöttek, mint például a *HBase* (oszlop-orientált adatbázis), a *Zookeeper* (elosztott rendszer karbantartására szolgáló service) vagy például a *Mahout* (skalázható machine learning library).



4. ábra - Hadoop Ecosystem

A legkülönbözőbb felhasználói igények kielégítésére több Apache projekt is létrejött, melyek az alap Hadoop rendszer stabilitására és skalázhatóságára épülnek.

3. BioGrid Hadoop

Az alkalmazás legfőbb célja: biológiai gének egyedi és összetett interakcióinak a minél hatékonyabban és gyorsabban történő lekérdezése és rendezése. A gének kinyeréséhez a *BioGRID* nevű adatbázis *REST* alapú webszolgáltatását használja, amely adatbázis az egyik legnagyobb gyűjteménye a fehérje-fehérje és genetikai interakcióknak. (A BioGrid Hadoop megnevezés nem összetévesztendő a BioGRID adatbázissal és a Hadoop rendszerével, a két név csupán a kutatás könnyű azonosíthatósága miatt lett felhasználva.) A program grafikus felületen keresztül segít az eligazodásban, így használatához nem szükséges a Hadoop mélyebb ismerete. Csupán 3 dolgot kell megadni:

- lekérdezni kívánt gének listáját,
- kimeneti mappa helyét, ahol az output fájlt szeretnénk tárolni,
- Hadoop főkönyvtárának elérési útvonalát.

Ezen paraméterek ismeretében a program automatikusan fut, már csak a végeredményre (output file-ra) kell várnunk. A kimeneti fájl minden futás esetében egy *.csv* (comma-separated values) kiterjesztésű szöveges állomány lesz, ami egy táblázatot reprezentál, ahol az egyes mezőértékek vesszővel (vagy más karakterrel) vannak elválasztva. (Bővebb leírás CSV-ről a *custom.jar* részben található.) Ezt a formátumot a táblázatkezelő programok nagy része kiválóan tudja importálni és exportálni.

3.1 Felhasznált eszközök

3.1.1 Operációs Rendszer

Ahhoz, hogy a BioGrid Hadoop megszülethessen, első sorban egy megfelelő fejlesztőkörnyezetre volt szükségem. Választásom az Ubuntu Linux 12.04-re esett, amelyhez a Hadoop legfrissebb verzióját telepítettem fel. (2013 augusztusában ez az 1.2.1 (stable) verzió volt.) Konfigurálható rendszer lévén a Hadoop képes a "single node"-ként való üzemeléshez. Ehhez csupán annyi szükséges, hogy a hivatalos oldalról (<http://hadoop.apache.org/releases.html>) letöltött *tar.gz* állományt kicsomagoljuk, a Java 7-es verzióját feltelepítjük, majd a környezeti változókat a *profile options*-el egyetembe frissítjük.

Így a rendszer "standalone" módban már működőképpessé is vált, viszont ilyenkor az egész rendszer működésért csak egy Java processz felelős, tehát a helyi rendszerünkön csak egy JVM (Java Virtual Machine) instance fog futni. A HDFS szolgáltatásokat sajnos így nem érthetjük el, sőt külön a Tracker-eket sem kísérhetjük nyomon. Fejlesztés szempontjából ez a mód éppen megfelelő is lehetett volna, de a Hadoop skálázhatóságának demonstrálása miatt "pseudo-distributed" módban akartam használni a rendszert, így magam konfiguráltam be az egyes komponenseket. Három xml fájlt módosítottam, a *core-site.xml*-t, a *hdfs-site.xml*-t és a *mapred-site.xml*-t.

Minden ilyen konfigurációs fájl a Hadoop főkönyvtárának *conf* almappjában található meg. A *core-site.xml*-ben a NameNode host-ját local-ra, a kommunikációs portját pedig 9000-re állítottam:

core-site.xml:

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

A *hdfs-site.xml*-ben a replikációs tényezőt 1-re (az egy node-on való fejlesztés szempontjából nem tartottam fontosnak a redundanciát), végül a *mapred-site*-ban a Job Tracker futását a saját gépem 9001-es portjára konfiguráltam be:

hdfs-site.xml:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

mapred-site.xml:

```
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:9001</value>
  </property>
</configuration>
```

Rendszer telepítésének befejezése a NameNode formázásával zárult, amelyhez a *bin/hadoop namenode -format* parancsot használtam. Ez lényegében az összes NameNode-on található metaadatot törölte, majd a beállításaim szerint alaphelyzetbe állította a csomópontot. Ezek után már csak a rendszer elindítása volt hátra, az ehhez pedig a főkönyvtár

`bin` mappájában található `start-all.sh` script-et futtattam le. Ezután, a `jps` (Java Process Status) paranccsal máris láthatóvá vált, hogy a Task Tracker, NameNode, DataNode és Job Tracker elindult, így a rendszer üzemképessé vált.

A Job-ok és a HDFS monitorozását a Hadoop a webes felületein, a `http://localhost:50030` és `50070`-es címeken keresztül végeztem. Itt tudtam pontosan nyomon követni a futási időket, azt hogy mennyi memóriát, tárhelyet és használ egy adott job, valamint ha valami rosszul futott le, az adott job-hoz tartozó log-okat is ezen keresztül tudtam böngészni.

Hadoop job_201410241045_0001 on <u>localhost</u>							
User: jw Job Name: streamjob7804119538111868898.jar Job File: hdfs://localhost:9000/tmp/hadoop-jw/mapred/staging/jw/staging/job_201410241045_0001/job.xml Submit Host: jw-VirtualBox Submit Host Address: 127.0.1.1 Job-ACLs: All users are allowed Job Setup: Successful Status: Succeeded Started at: Fri Oct 24 10:54:47 CEST 2014 Finished at: Fri Oct 24 10:55:31 CEST 2014 Finished in: 43sec Job Cleanup: Successful							
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	2	0	0	2	0	0 / 0
reduce	100.00%	1	0	0	1	0	0 / 0

	Counter	Map	Reduce	Total
File Input Format Counters	Bytes Read	0	0	56
Job Counters	SLOTS_MILLIS_MAPS	0	0	48,122
	Launched reduce tasks	0	0	1
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Launched map tasks	0	0	2
	Data-local map tasks	0	0	2
	SLOTS_MILLIS_REDUCE	0	0	13,577
File Output Format Counters	Bytes Written	0	0	142,002

5. ábra - Információk egy MR job-ról
 A webes felületen az olvasott bájtok számától kezdődően, a MR task-ok státuszain át, egészen a Job futásának végéig minden folyamatot nyomon követhetünk.

3.1.2 Oracle VM Virtual Box

A saját elosztott környezetem kialakításához az Oracle VM Virtual Box virtualizációs szoftvert vettem segítségül. Az alkalmazás lehetővé tette számomra azt, hogy a feltelepített

operációs rendszeremen (*Windows 7*) felül olyan ún. vendég operációs rendszert (*Guest OS-t*) hozhassak létre, amely az általam konfigurált virtuális környezet erőforrásait használva volt képes működni. A Guest OS az én esetemben Ubuntu Linux volt, amely az előző részben már említésre került. A rendszerinformációkat az alábbi táblázat tartalmazza:

OS:	<i>Ubuntu Linux 12.04.4 LTS</i>
CPU:	<i>Intel Core i3-380 (2.53 GHz)</i>
Utasításkészlet:	<i>64 Bit</i>
Memória:	<i>4096 MB</i>
Háttértár:	<i>20,66 GB</i>

*1. táblázat - A virtuális környezet paraméterei
Operációs rendszerként az Ubuntu Linux disztribúció 64 bites változatát használtam, ami alatt egy kétmagos Intel i3-as processzor működött 4GB RAM és 20 GB háttértár kíséretében. Fejlesztési szempontból ez a konfiguráció igen hatékonynak bizonyult.*

3.1.3 Hadoop Streaming

Minden Hadoop rendszer installálásával párhuzamosan települ a *Hadoop Streaming* nevű utility, amely lehetővé teszi azt, hogy bármilyen futtatható script-et mapper-ként vagy reducer-ként használjunk egy job-nál. Alapból a Java nyelven kódolt MR programok külön osztályokat használnak az adatok HDFS-ről való beolvasására és kiíratására, a Streaming viszont azok helyett inkább az *stdin*-t és az *stdout*-ot használja. Ez könnyebbséget jelent, hiszen így lényegében bármilyen nyelven tudunk Big Data programokat írni, mivel a nyelvek jelentős része képes ezzel a két I/O stream-el kommunikálni. Ezt kihasználva az alkalmazásom végül Python nyelven lett kódolva, viszont hogy a kimeneti fájl .csv formátumú lehessen, egy kevés Java kód is szükséges volt hozzá.

A utility működése nagyon egyszerű. Egy parancs, néhány kapcsoló, amelyek közül mi magunk választjuk ki, hogy mire is van valójában szükségünk. A BioGrid Hadoop esetében például a MR job futtatásához szükséges parancs az alábbiak szerint néz ki:

```
$ bin/hadoop jar contrib/streaming/hadoop-streaming-1.2.1.jar \  
-file /home/jw/python/mapper.py \  
-mapper /home/jw/python/mapper.py \  
-file /home/jw/python/reducer.py \  
-reducer /home/jw/python/reducer.py \  
-input /user/jw/* \  
-output /user/jw/output
```

Itt látható a mapper-nek és a reducer-nek, valamint a HDFS-en használatos input/output mappáknak az elérési útvonalai. További opciók is vannak ezeken kívül, akár még Partitioner osztályt is megadhatunk, ha a programunk azt kívánja. A fejlesztéshez kicsit körülményes volt mindig felparaméterezni egy Streaming job-ot, ezért sokszor csak az alábbi parancsot alkalmaztam emulálásra:

```
$ cat input.txt | ./mapper.py | sort | ./reducer.py
```

Mivel az alkalmazás az stdin-t és stdout-ot használja, ez az egyszerű Unix parancs nagyon hasznosnak bizonyult a BioGrid Hadoop fejlesztése alatt.

3.1.4 A Python és az EasyGUI

A mapper-t, reducer-t és a GUI-t egyaránt a manapság nagy népszerűségnek örvendő Python nyelven kódoltam. A Python a nagyon magas programozási nyelvek csoportjába tartozó nyelv, melynek filozófiája az olvashatóságot és a programozás megkönnyítését tartja fontosnak, szemben például a futási idővel. Többféle programozási paradigmát támogat (funkcionális, objektumorientált, procedurális), valamint MapReduce programok írására is könnyen használható. Az alkalmazásomban csupán alapvető modulokkal dolgoztam, melyek közül csak az EasyGUI volt kivétel.

A grafikus felület programozásakor nem a design-t, hanem a könnyű kezelhetőséget tartottam a legfontosabbnak. Egy Hadoop Streaming job felparaméterezésekor sok mindenre kell ügyelni és figyelni, ezért fontosnak tartottam azt, hogy a programot egy alap Unix-tudással rendelkező felhasználó is egyszerűen tudja kezelni. A lehetőségeket böngészve az EasyGUI mellett tettem le a voksomat, mivel a többi GUI-val ellentétben ez a modul nem eseményvezérelt (*event-driven*) elven működik, tehát a dialógusablakokat egyszerű függvényhívásokon keresztül tudtam létrehozni. (Nem volt szükséges se *root window*-t deklarálni, se *event loop*-ot indítani, mint például a *Tkinter* esetében.)

3.2 A Repository bemutatása

Az alkalmazás verziókezeléséhez *git*-et használtam, melynek repository-ja az alábbi linken érhető el: <https://github.com/benauer/Biogrid-Hadoop/>. A program használatához a *Hadoop* 1.2.1 és a *Python* 2.x-es verziója (az EasyGUI modullal kiegészülve) szükséges.



6. ábra BioGRID és a Hadoop logója

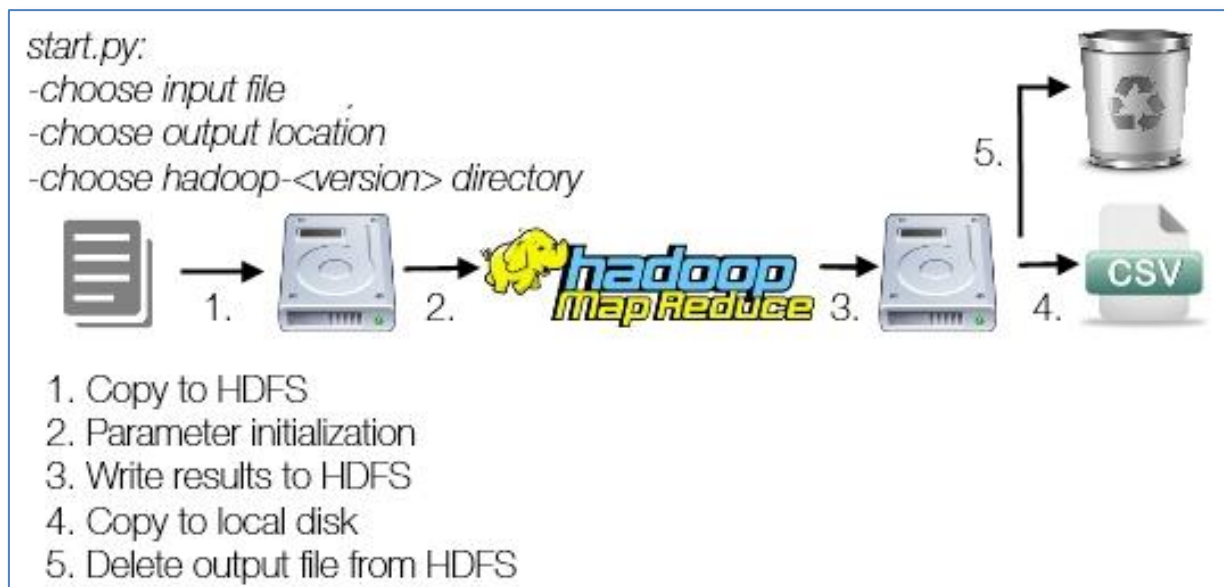
A gének listáját, melyek interakciót le szeretnénk kérdezni, egy *.txt* fájlban szükséges megadni méghozzá úgy, hogy egy gén egy sorban szerepeljen:

***input.txt* (példa):**

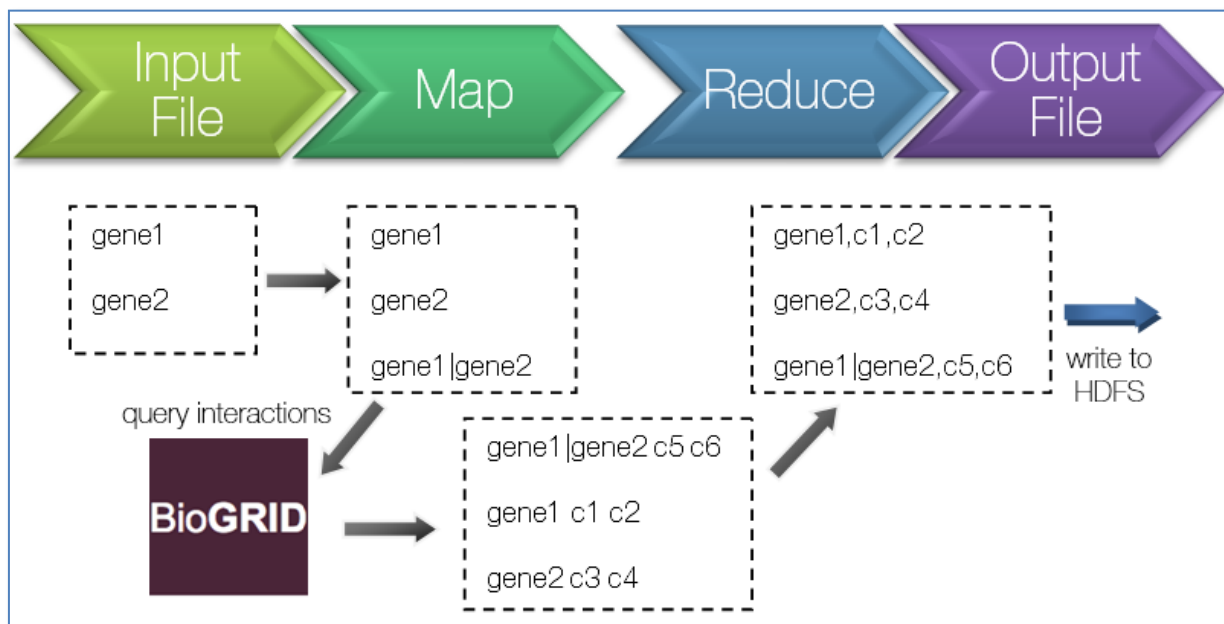
```
CG13321
CG13322
CG13326
CG14247
Glycogenin
CG11796
CG12582CG13321
```

Output mappának (a kimeneti fájl tárolásához) tetszőlegesen választható bármilyen hely, ahova van írási jogosultsága a felhasználónak, a Hadoop alapkönyvtárhoz pedig (ha a környezeti változók be vannak állítva) a *\$HADOOP_INSTALL* alatti *hadoop-<verziószám>* mappát szükséges beállítani. (Tehát a dialógus ablakon keresztül azt a mappát kell megnyitni.) A futás egyes fázisait nyomon követhetjük a terminálablakon keresztül, a folyamat végét pedig egy külön ablak fogja számunkra jelezni. A kimeneti fájlt ekkor már használhatjuk is, importálhatjuk táblázatkezelőbe vagy akár egy egyszerű notepad-el is megnyithatjuk.

A program sematikus működését a 7. és 8. ábra szemlélteti:



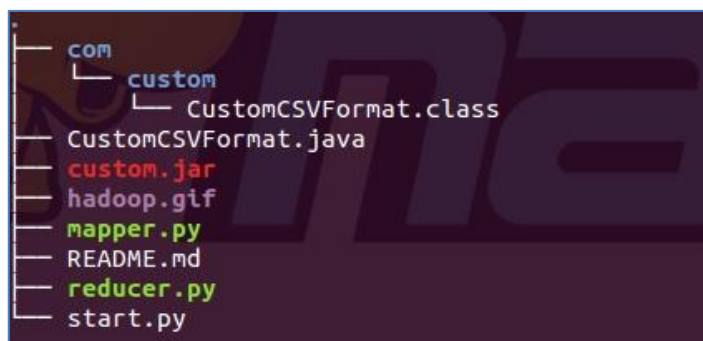
7. ábra - A program működése. A program a megfelelő paramétereinek beállítása után a bemeneti fájlt a HDFS-re lesz felmásolva, hiszen a MR job majd onnan fogja beolvasni a fájlt. A job végeztével a kimeneti fájlt a saját lokális fájlrendszerünkre kerül, a HDFS pedig a kiinduló állapotába tér vissza.



8. ábra - A MapReduce folyamata

A bemeneti fájlként megadott két génből előállítható részhalmazokat a Map folyamata hozza létre, melyek a BioGRID adatbázison keresztüli lekérdezésekhez szükségesek. Az interakciókat a program számára egy REST service biztosítja, az adatok rendezését és csv-vé való alakítását pedig a Reducer task végzi el. A job a HDFS-re való írással fejeződik be.

A könyvtárszerkezet az alábbiak szerint épül fel (9. ábra):



9. ábra - Könyvtárszerkezet, amit a repository klónozása után kapunk

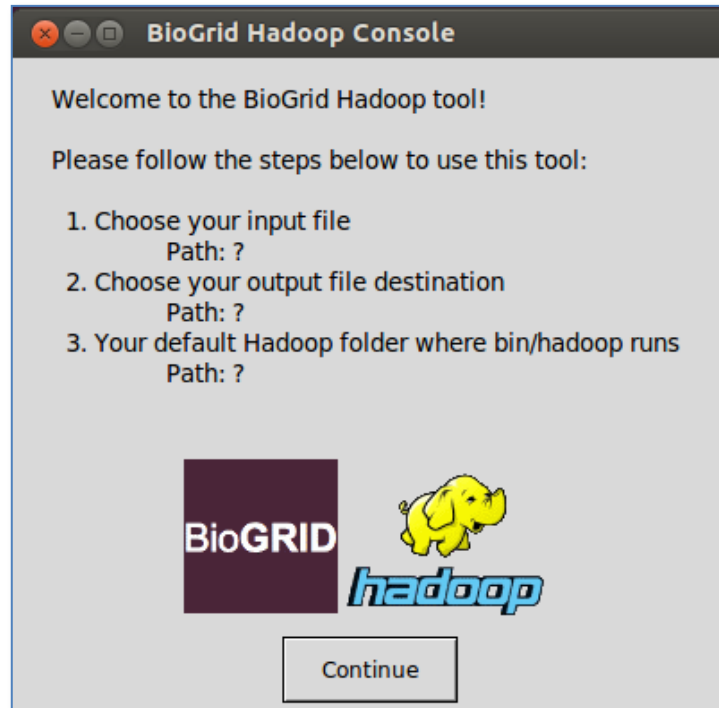
3.2.1 GUI (start.py)

A program a MR job levezényléséért, az ehhez szükséges paraméterek bekéréséért és az egyszerű használatért felelős. A felhasználónak nem kell külön foglalkozni a HDFS-re való felöltéssel és az onnan való letöltéssel, csupán a grafikus interface-en keresztül szükséges megadnia néhány elérési útvonalat. A kód elején – bármely programhoz hasonlóan – az importok helyezkednek el. Az *os*, *getpass* és *subprocess* modulokat az operációs rendszer-függő parancsok használata-, az *easygui*-t pedig az egyszerű grafikus kezelőfelület létrehozása miatt importáltam be.

A main függvény eleje az üdvözlőablak megjelenítésével kezdődik (10. ábra), ami közli velünk, hogy milyen paraméterek szükségesek a géninterakciók létrehozásához. Ehhez az *easygui*-nak a *msgbox* függvényét használtam, amely a szöveg mellett a projekt logóját is megjeleníti („*hadoop.gif*”).

start.py (1)

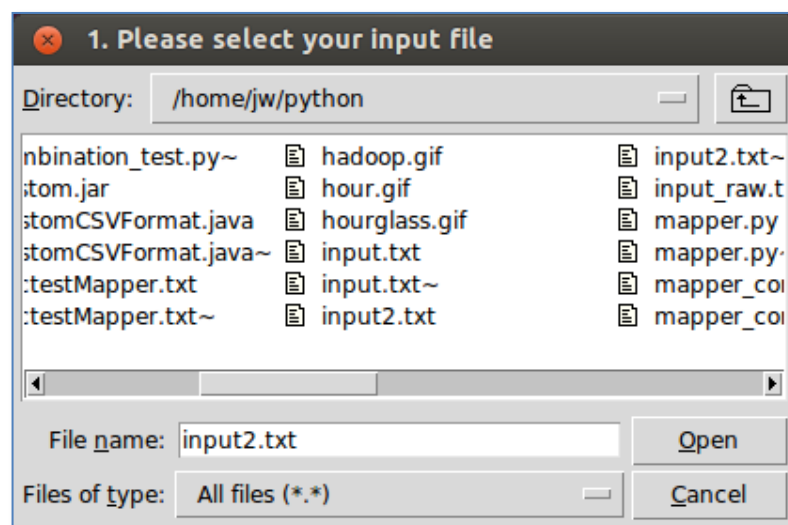
```
10. image="hadoop.gif"
11. msgbox(msg="Welcome to the BioGrid Hadoop tool!\n\nPlease follow the
    steps below to use this tool:\n\n 1. Choose your input file\n\tPath: ?\n 2. Choose your output file destination \n\tPath: ?\n 3. Your default H
    adoop folder where bin/hadoop runs \n\tPath: ?\n", title='BioGrid Hadoop
    Console', ok_button='Continue', image=image)
```



10. ábra - A szoftver kezdőképernyője, amely a programon belüli tájékozódást segíti.

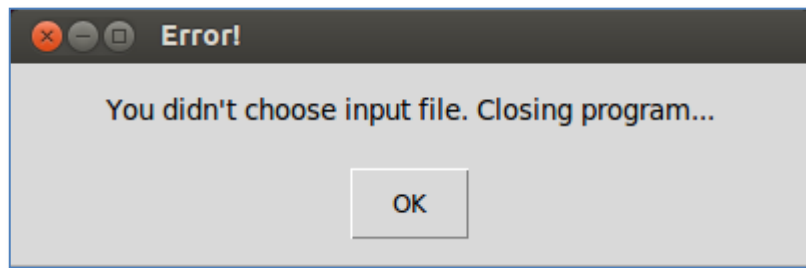
A *title* paraméterrel az ablak nevét, az *ok_button*-al pedig a gomb feliratát állítottam be.

Az input fájl és az output mappa kiválasztását a *fileopenbox* végzi el, amely egy dialógus ablakon keresztül segít az egyszerű fájl- és mappakiválasztásban. (11. ábra) Az input fájlban egy génnek egy sorban kell elhelyezkednie, mivel az MR kód eszerint fogja majd bontani az egyes task-okat. Output file útvonalnak pedig egy tetszőleges mappát kell megadni a saját, lokális rendszerünkön.



11. ábra - Az input fájl és az output könyvtár kiválasztását segítő dialógusablak

(Alapból ez az aktuális mappa lesz, ahonnan futtatjuk az alkalmazást (`default=''`).) Egy kisebb hibakezelés is belekerült a két dialógus közé, ha nem lett kiválasztva egy paraméter, akkor a program be fog zárulni majd 1-es hibakóddal fog visszatérni. (12. ábra)

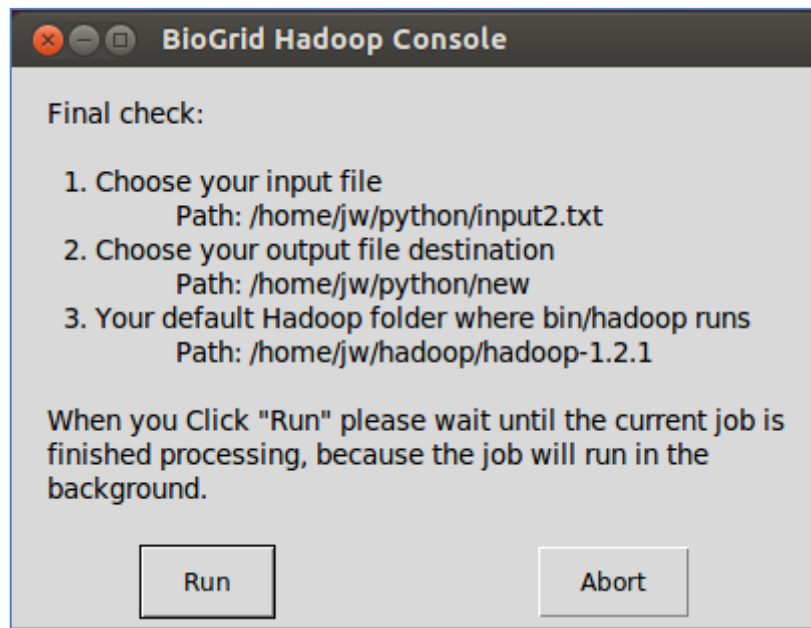


12. ábra - Hibaüzenet-ablak.
Ha valamilyen útvonalat nem állítottunk be helyesen, akkor a program hibaüzenetet fog megjeleníteni.

start.py (2)

```
13. #Get information about input/output paths and hadoop
14. inputFile = fileopenbox(msg=None, title="1. Please select your input
    file", default='*', filetypes=None)
15.
16. if inputFile == None:
17.     msgbox(msg="You didn't choose input file. Closing program...", ti
        tle='Error!', ok_button='OK', image=None)
18.     print "You didn't choose input file. Closing program..."
19.     return 1
20.
21. #Choose output directory
22. msgbox(msg="Please follow the steps below to use this tool:\n\n 1. C
    hoose your input file\n\tPath: " + inputFile + "\n 2. Choose your output
    file destination \n\tPath: ?\n 3. Your default Hadoop folder where bin/h
    adoop runs \n\tPath: ?\n", title='BioGrid Hadoop Console -
    Step 2', ok_button='Continue', image=image)
23.
24. outputFileDir = diopenbox(msg=None, title="2. Please select your out
    put folder", default='*')
25.
26. if outputFileDir == None:
27.     msgbox(msg="You didn't choose output file. Closing program...", t
        itle='Error!', ok_button='OK', image=None)
28.     print "You didn't choose output file. Closing program..."
29.     return 1
```

Ehhez hasonlóan, végül a Hadoop könyvtárát kell megadni, ahonnan a `bin/hadoop` kezdetű parancsokat lehet kiadni. Majd végül egy végső ellenőrzés következik, ahol akár el is tudjuk „abort”-álni a program futását. (13. ábra)



13. ábra - A MR job indítása előtt egy végső "Final check" ablak jelenik meg, ahol ellenőrizni tudjuk a beállításainkat.

start.py (3)

```

31.     #Choose Hadoop directory
32.     msgbox(msg="Please follow the steps below to use this tool:\n\n 1. C
choose your input file\n\tPath: " + inputFile + "\n 2. Choose your output
file destination \n\tPath: " + outputFileDir + "\n 3. Your default Hadoo
p folder where bin/hadoop runs \n\tPath: ?\n", title='BioGrid Hadoop Cons
ole - Step 3', ok_button='Continue', image=image)
33.
34.     hadoopDefaultDir = diropenbox(msg="where bin/hadoop runs", title="3.
Default hadoop location", default="/home")
35.
36.     if hadoopDefaultDir == None:
37.         msgbox(msg="You didn't specify your Hadoop folder. Closing progra
m...", title='Error!', ok_button='OK', image=None)
38.         print "You didn't specify your Hadoop folder. Closing program..."
39.         return 1
40.
41.     #Final check
42.     choice = buttonbox(msg="Final check:\n\n 1. Choose your input file\n
\tPath: " + inputFile + "\n 2. Choose your output file destination \n\tPa
th: " + outputFileDir + "\n 3. Your default Hadoop folder where bin/hadoo
p runs \n\tPath: " + hadoopDefaultDir + "\n\nWhen you Click \"Run\" pleas
e wait until the current job is finished processing, because the job will
run in the background.", title='BioGrid Hadoop Console', choices=["Run",
"Abort"])
43.
44.     if choice == "Abort":
45.         return 1

```

A HDFS-re való feltöltést `bin/hadoop dfs -copyFromLocal`-al lehet végrehajtani unix shell-ből (bash). Minden ilyen parancshoz a programon belül a `subprocess` modul `call` függvénye lett meghívva.

start.py (4)

```
52. subprocess.call([copyInputFile, "dfs", "copyFromLocal", inputFile, "input/input.txt"])
```

Ezek után a Hadoop Streaming futásához szükséges kapcsolóknak a definiálása következik (sztringként). Az eredeti parancs alapján,

From terminal:

```
$ bin/hadoop jar contrib/streaming/hadoop-streaming-1.2.1.jar \  
-libjars /home/jw/python/custom.jar \  
-file /home/jw/python/mapper.py \  
-mapper /home/jw/python/mapper.py \  
-file /home/jw/python/reducer.py \  
-reducer /home/jw/python/reducer.py \  
-outputformat com.custom.CustomCSVFormat \  
-input /user/jw/* -output /user/jw/output/
```

a változók az alábbiak szerint lettek deklarálva:

start.py (5)

```
54. #Initialization of parameters  
55. mapper = os.getcwd() + "/mapper.py"  
56. reducer = os.getcwd() + "/reducer.py"  
57. customJar = os.getcwd() + "/custom.jar"  
58. inputPath = "/user/" + getpass.getuser() + "/*"  
59. outputPath = "/user/" + getpass.getuser() + "/output/"  
60. os.chdir(hadoopDefaultDir)
```

Az Unix parancsnál kiadott kapcsolók a Hadoop Streaming rövid leírásánál megtalálhatóak, kivéve az `-outputformat`. Nevéből következtetve a kimeneti fájlunk kiterjesztését lehet vele meghatározni. Az `os.chdir`-el a Hadoop könyvtárba navigál el a program, hogy a `bin/hadoop`-os Streaming job futásához szükséges parancsot ki tudjuk adni.

Ezután a job-nak az elindítása következik, ehhez szintén - a HDFS-re való másoláshoz hasonlóan - a `subprocess.call`-t használtam:

start.py (6)

```
66. subprocess.call(["bin/hadoop", "jar", "contrib/streaming/hadoop-streaming-1.2.1.jar", "-libjars", customJar, "-file", mapper, "-mapper", mapper, "-file", reducer, "-reducer", reducer, "-outputformat", "com.custom.CustomCSVFormat", "-input", inputPath, "-output", outputPath])
```

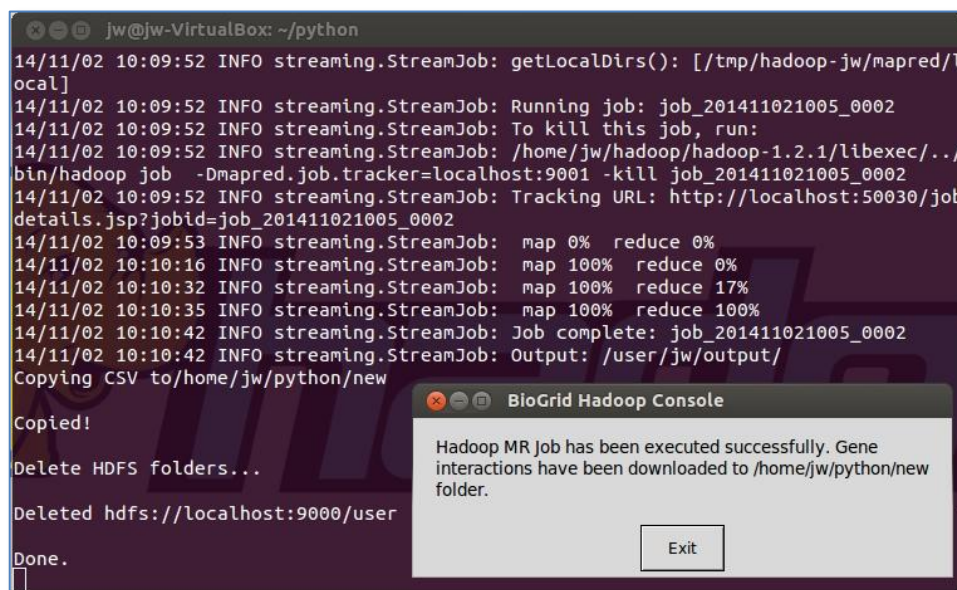
Ez a folyamat az általunk megadott input fájl méretétől- és a cluster-ünk nagyságától függően akár néhány percre is eltarthat. (A saját single-node-os rendszeremen, 10 génre rendszerint kb. 1 perc volt a futási idő.) Amint a job lefutott, a kimeneti fájlunk a megadott output folder-be lesz másolva majd a HDFS-ről az általunk használt mappák törlése következik,

start.py (7)

```

69.     print "Copying CSV to" + outputFileDir + '\n'
70.     outputPath = "/user/" + getpass.getuser() + "/output/*.csv"
71.     error = subprocess.call(["bin/hadoop", "dfs", "-
    get", outputPath, outputFileDir])
72.
73.     if error == 0:
74.         print "Copied!\n"
75.     else:
76.         print "ERROR in command get...\n"
77.
78.     #Delete HDFS
79.     print "Delete HDFS folders...\n"
80.     error = subprocess.call(["bin/hadoop", "dfs", "-rmr", "/user/"])
81.     if error != 0:
82.         print "ERROR!\n"
83.
84.     print "\nDone."
```

egy esetleges új MR job hibátlan futásának érdekében. (Lényegében a HDFS-en lévő /user mappát rekurzívan kitörli.) Végül - ha nem volt semmilyen hiba – egy ablak jelenik meg, amely az alkalmazás futásának végét jelzi. (14. ábra)



14. ábra - Az MR job futásának végét jelző ablak. A job részletes futását a terminál ablakon keresztül tudjuk leginkább nyomon követni.

start.py (8)

```
86.     #Exit message
87.     msgbox(msg="Hadoop MR Job has been executed successfully. Gene interactions have been downloaded to " + outputFileDir + " folder.", title='Bio Grid Hadoop Console', ok_button='Exit')
88.
89.     return 0
```

3.2.2 Mapper (mapper.py)

Amikor a Streaming-hez szükséges parancs kiadásra kerül, a utility az adott elérési útvonalon - ahonnan a start.py-t is futtatva van - keresni fogja a mapper.py programot, amelynek az egyik legfontosabb feladata a géninterakcióknak a lekérdezése. A kód az alábbiak szerint néz ki:

mapper.py

```
1.  #!/usr/bin/env python2
2.
3.  from itertools import chain, combinations
4.  import sys
5.  import urllib2
6.  import subprocess
7.  import os
8.
9.  #Function to produce combinations of genes
10. def powerset(iterable):
11.     combList = list(iterable)
12.     return chain.from_iterable(combinations(combList,n) for n in range(len(combList)+1))
13.
14. def main():
15.     gens = sys.stdin.readlines()
16.     gens = [item.strip() for item in gens]
17.     gens = list(powerset(gens))
18.
19.     #For Biogrid REST service we need to separate gens like "gene1|gene2|gene3"
20.     for i in range(0, len(gens)) :
21.         gens[i] = "|".join(str(x) for x in gens[i])
22.
23.     #REST calls
24.     for i in range(1, len(gens)):
25.         url = 'http://webservice.thebiogrid.org/interactions/?searchNames=true&geneList=' + str(gens[i]) + '&taxId=all&accesskey=99380a201fcc2245c18f6e4d1fec9035'
```

```

26.         result = urllib2.urlopen(url).readlines()
27.
28.     #Merging of gene names and REST answers
29.     for j in range(0, len(result)):
30.         resultStr = "".join(str(gens[i]) + '\t' + result[j])
31.         sys.stdout.write(resultStr)
32.
33. if __name__ == "__main__":
34.     main()

```

A program main függvényében először a soronkénti olvasás folyamata zajlik le (`readlines()`), majd a felesleges whitespace karakterek kerülnek eltávolításra. A *gens*-re ezután a *powerset* függvény lesz meghívva, amely egy olyan listát alakít ki, amely az interakciók lekérdezésének alapját biztosítja. A függvény tulajdonképpen a géneket tartalmazó véges halmaznak (melynek elemei a bemeneti fájl egyes sorai,) a részhalmazait állítja elő. Például 4 darab gén esetén:

{1};{2};{3};{4} {1, 2};{1, 3};{1, 4};{2, 3};{3, 4};{2, 4} {1, 2, 3};{1, 2, 4};{1, 3, 4};{2, 3, 4}; {1, 2, 3, 4}	$2^n - 1 = 2^4 - 1 = 15$
--	--------------------------

Ahhoz hogy három gén interakcióját le lehessen kérdezni, a BioGRID REST leírása alapján a géneket "|" jellel kell elválasztani ("gén_1|gén_2|gén_3").

First 10 interactions for human MDM2 gene:

[http://webservice.thebiogrid.org/interactions/?searchNames=true&geneList=MDM2&includeInteractors=true&taxId=9606&start=0&max=10&accesskey=\[ACCESSKEY\]](http://webservice.thebiogrid.org/interactions/?searchNames=true&geneList=MDM2&includeInteractors=true&taxId=9606&start=0&max=10&accesskey=[ACCESSKEY])

Interactions between S. cerevisiae CDC27, APC1 and APC2:

[http://webservice.thebiogrid.org/interactions/?searchNames=true&geneList=cdc27|apc1|apc2&taxId=559292&accesskey=\[ACCESSKEY\]](http://webservice.thebiogrid.org/interactions/?searchNames=true&geneList=cdc27|apc1|apc2&taxId=559292&accesskey=[ACCESSKEY])

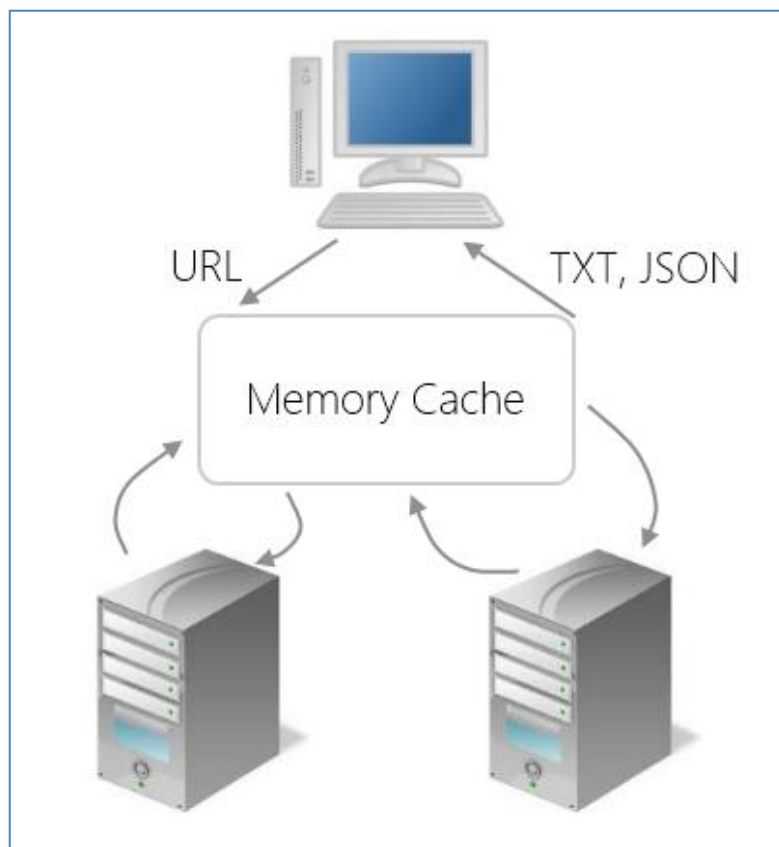
Interactions from pubmed IDs 18316726 and 17662948:

[http://webservice.thebiogrid.org/interactions/?pubmedList=18316726|17662948&accesskey=\[ACCESSKEY\]](http://webservice.thebiogrid.org/interactions/?pubmedList=18316726|17662948&accesskey=[ACCESSKEY])

15. ábra - A CDC27, APC1 és APC2 génhez szükséges link felépítése. A BioGRID REST szolgáltatása több opciót is felkínál az interakciók lekérdezésére.

Ezt a műveletet a 23. sor *for* ciklusa végzi el. A linkeket az ezután következő ciklus állítja elő, méghozzá úgy hogy a második elemtől indítva (mivel az első halmaz üres) a már előállított ("|" jellel elválasztott) gének sztringjeit a cím megfelelő helyére szúrja be.

A címek a lekérdezésekhez szükségesek, amit BioGRID a REST szoftverarchitektúrán keresztüli webservice-el biztosítja. Konceptióját tekintve egyszerű kliens-szerver architektúrának tekinthető, ahol a kliensek kéréseket indítanak a szerver felé, majd miután a szerver feldolgozta azt, a megfelelő válasz kerül visszaküldésre.



16. ábra - REST architektúra

A BioGRID adatbázisból egy egyedi linken keresztül tudjuk lekérdezni a kívánt interakciókat, melyek egyszerű txt vagy JSON formátumúak lehetnek.

A kérések és válaszok erőforrásreprezentációk szállítása köré épülnek. Erőforrás lényegében bármi lehet, ami összefüggő és értelmesen címezhető entitás. Tipikusan egy dokumentumot tekinthető erőforrásnak, a BioGRID esetében pedig olyan egyéni txt vagy JSON állományokat, amelyek a géninformációkat tartalmazzák. Ezeket egy egységes azonosító - URI (Uniform Resource Identifier) - segítségével lehet lekérdezni, ami egy linken keresztül érhető el, melyet a REST szerver eléréséhez biztosít a webservice. Hogy használni lehessen, egy egyedi azonosítót szükséges igényelni a BioGRID weboldalán keresztül, majd azt beleépítve egy linkbe lehet saját kéréseket indítani. (A projekthez külön lett igényelve ilyen azonosító, ez a programkódban is megtalálható.)

A programban ezután következik maga a hívási rész, ahol az `urllib2` modul `urlopen` függvénye soronként olvassa be a REST szerver által küldött válaszokat. Ahhoz, hogy tudni lehessen, melyik kimenet melyik interakcióhoz tartozik, minden kimeneti sornak az elejére odakerült, hogy melyik gén(ek)ről van szó pontosan. A webservice-től érkezett interakciók mezőit pedig a program tabulátorral fogja elválasztani és az így kapott sztringeket fogja majd az stdout-ra írni.

3.2.3 Reducer (`reducer.py`)

A program ezen része a mapper-nél előállított köztes értékek rendezéséért felelős. Egy single node-os rendszeren ez a folyamat kevésbé fontos, mint például egy több száz (esetleg több ezer) node-os rendszeren, ugyanis nem tudhatjuk biztosra azt, hogy az összetartozó gének egy helyen lettek-e lekérdezve.

`reducer.py`:

```
1. #!/usr/bin/env python2
2.
3. import sys
4. import subprocess
5. def main():
6.     #Replacing tabs to commas
7.     for line in sys.stdin:
8.         line = line.replace('\t', ',')
9.         sys.stdout.write(line)
10.
11. if __name__ == "__main__":
12.     main()
```

Az alkalmazás szempontjából ez a kód csupán egy dologért felel a rendezésen kívül: a megkapott interakció-információ mezők közötti tabulátoroknak a vesszőre való kicseréléséért. Ezt - a mapper-től kapott összes soron végighaladva -, a `line.replace` függvény végzi el. A kimenetet az előző programhoz hasonlóan az stdout-ra írja ki, amit a Streaming utility a HDFS-re való írással fog befejezni.

3.2.4 `custom.jar` (`CustomCSVFormat.java`)

A Hadoop Streaming-nek a dokumentációit bújva egy igen érdekes dolgot véltem felfedezni: akár saját kiterjesztéssel is elláthatjuk egy job-nak a kimeneti fájljait. A

használhatóságot szem előtt tartva úgy döntöttem, hogy az interakciók tárolásának a legmegfelelőbb formátuma a CSV lesz.

A CSV (comma-separated values) "vesszővel elválasztott értékeket" jelent. Más helyeken szokták character-separated values-nak ("karakterrel elválasztott értékeknek") is hívni, mivel az elválasztó karakternek nem feltétlenül kell vesszőnek lennie, hanem bármely más whitespace karaktert is választható helyette. A reducer-nél a táblázat is megfelelő lehetett volna, viszont az *RFC 4180* alapján, inkább a vesszőt választottam elválasztó karakterként. Valójában az egész fájl egy táblázatnak tekinthető, amelyben a rekordok sorzáró karakterrel az egyes mezőértékek pedig egy mindenhol azonos, de a nyers szövegben nem előforduló karakterekkel vannak elszeparálva. CSV-t a legtöbb irodai táblázatkezelő, business-orientált és tudományos alkalmazás támogatja. Ezek mellett nem meglepő módon az *RDBMS*-ek nagyrésze is képes kezelni. A formátumnak hivatalos standard-je nincs, csupán a fent említett RFC dokumentum foglalkozik vele a szokásosnál kicsit mélyrehatóbban. A mezőértékek a legtöbb karakterkódolást támogatják (*ASCII*, *UTF-8*), ezen kívül minden rekordnak ugyanannyi mezőértéket kell tartalmaznia. (Azaz minden sorban ugyanannyi vesszőnek kell lennie.)

	A	B	C	D	E	F	G	H	I	J	K
1	CG11790	31191	35279	42999	61251	67920	Dmel CG10664	Dmel CG11790	CG10664	CG11790	COXIV Dmel CG10664 cox4 IV COX IV
2	CG11790	31769	42999	42731	67920	67676	Dmel CG11790	Dmel CG6733	CG11790	CG6733	Dmel CG11790
3	CG11790	31768	42999	41454	67920	66571	Dmel CG11790	Dmel CG3397	CG11790	CG3397	Dmel CG11790
4	CG11790 RNF11	29260	26994	2060	117941	108374	CGI-123		0 RNF11	EPS15	SID1669
5	CG11790 RNF11	932011	26994	7316	117941	113164	CGI-123		0 RNF11	UBC	SID1669
6	CG11790 RNF11	928130	26994	2060	117941	108374	CGI-123		0 RNF11	EPS15	SID1669
7	CG11790 RNF11	928129	26994	10254	117941	115548	CGI-123		0 RNF11	STAM2	SID1669
8	CG11790 RNF11	928128	26994	9372	117941	114773	CGI-123		0 RNF11	ZFYVE9	SID1669
9	CG11790 RNF11	928127	26994	9372	117941	114773	CGI-123		0 RNF11	ZFYVE9	SID1669
10	CG11790 RNF11	893299	5859	26994	111797	117941		0 CGI-123	QARS	RNF11	GLNRS PRO2195
11	CG11790 RNF11	874469	26994	10213	117941	115508	CGI-123		0 RNF11	PSMD14	SID1669
12	CG11790 RNF11	874468	26994	5719	117941	111691	CGI-123		0 RNF11	PSMD13	SID1669
13	CG11790 RNF11	874467	26994	5718	117941	111690	CGI-123		0 RNF11	PSMD12	SID1669
14	CG11790 RNF11	874466	26994	5717	117941	111689	CGI-123		0 RNF11	PSMD11	SID1669
15	CG11790 RNF11	874465	26994	5716	117941	111688	CGI-123	RP5-889N15.3	RNF11	PSMD10	SID1669
16	CG11790 RNF11	874464	26994	5714	117941	111686	CGI-123		0 RNF11	PSMD8	SID1669
17	CG11790 RNF11	874463	26994	5713	117941	111685	CGI-123		0 RNF11	PSMD7	SID1669
18	CG11790 RNF11	874462	26994	9861	117941	115195	CGI-123		0 RNF11	PSMD6	SID1669
19	CG11790 RNF11	874461	26994	5711	117941	111684	CGI-123	RP11-27I1.5	RNF11	PSMD5	SID1669
20	CG11790 RNF11	874460	26994	5710	117941	111683	CGI-123	RP11-126K1.1	RNF11	PSMD4	SID1669
21	CG11790 RNF11	874459	26994	5709	117941	111682	CGI-123		0 RNF11	PSMD3	SID1669
22	CG11790 RNF11	874458	26994	5708	117941	111681	CGI-123		0 RNF11	PSMD2	SID1669
23	CG11790 RNF11	874457	26994	5707	117941	111680	CGI-123		0 RNF11	PSMD1	SID1669
24	CG11790 RNF11	874456	26994	63893	117941	121973	CGI-123		0 RNF11	UBE20	SID1669
25	CG11790 RNF11	874455	26994	7336	117941	113184	CGI-123		0 RNF11	UBE2V2	SID1669

17. ábra - Importált CSV fájl a LibreOffice Calc programban
Az ábrán a BioGrid Hadoopegyik lehetséges kimenete látható.

A kiterjesztés megváltoztatásához egy rövid Java kódot írtam, melyben a *MultipleTextOutputFormat* osztály *generateLeafFileName* metódusát írtam felül (*@Override*). Csupán annyit kellett módosítani, hogy visszatérési értéként a *name* paraméterhez a ".csv" kiterjesztést hozzáadtam. Gyakorlatilag ide bármilyen tetszőleges szöveget be lehetett volna írni és a Hadoop ezután az általunk megadott kiterjesztéssel látta volna el az összes kimeneti fájlunkat. (Összes kimeneti fájlról akkor beszélhetünk, ha például nincsen reducer-ünk. Ugyanis van lehetőségünk akár olyan Streaming MR job-ot is létrehozni, amelyben nincs reducer, ekkor a mapper általi köztes adatok nem lesznek rendezve egy fájlban, hanem az egyes részek külön-külön lesznek elérhetőek a HDFS-en.)

CustomCSVFormat.java:

```
1. package com.custom;
2. import org.apache.hadoop.fs.Path;
3. import org.apache.hadoop.io.Text;
4. import org.apache.hadoop.mapred.libMultipleTextOutputFormat;
5.
6. public class CustomCSVFormat extends MultipleTextOutputFormat<Text, Text
   > {
7.
8.     @Override
9.     protected String generateLeafFileName(String name) {
10.         return name + ".csv";
11.     }
12.
13. }
```

A Java forráskódot a *javac -cp* paranccsal fordítottam le, majd a Jar-ba történő csomagolását a *jar cvf*-el utasítással végeztem el. A fájlt, a *-Libjars* kapcsolóval lehet működésbe hozni a Streaming-en keresztül, ami a *start.py*-ban is megtalálható.

3.3 Tesztelés

Nagyméretű adatokról lévén szó, a szoftver fejlesztése után szükségesnek tartottam megvizsgálni a szoftver futásának kimeneti fájlait, a lekérdezni kívánt gének számának függvényében. A teszteléshez egy tetszőleges génlistát generáltam, melynek neveit és a hozzájuk tartozó lekérdezések számát a 2. táblázat tartalmazza. Célom az volt, hogy megtaláljam azt a határt, ameddig egy egyszerű PC erőforrásai is elegendőek a szoftver hatékony futtatásához.

Sorszám	Gén	Lekérdezések száma
1.	<i>CG11791</i>	1
2.	<i>CG11790</i>	3
3.	<i>RNF11</i>	7
4.	<i>CG13321</i>	15
5.	<i>CG13322</i>	31
6.	<i>CG13326</i>	63
7.	<i>CG14247</i>	127
8.	<i>CG11796</i>	255
9.	<i>CG12582</i>	511
10.	<i>Syt1</i>	1023
11.	<i>Syt4</i>	2047
12.	<i>Sac1</i>	4095
13.	<i>Cyp6a19</i>	8191
14.	<i>Cyp6a13</i>	16383
15.	<i>XPC</i>	32767

2. táblázat - A vizsgált gének listája és a hozzájuk tartozó lekérdezések száma

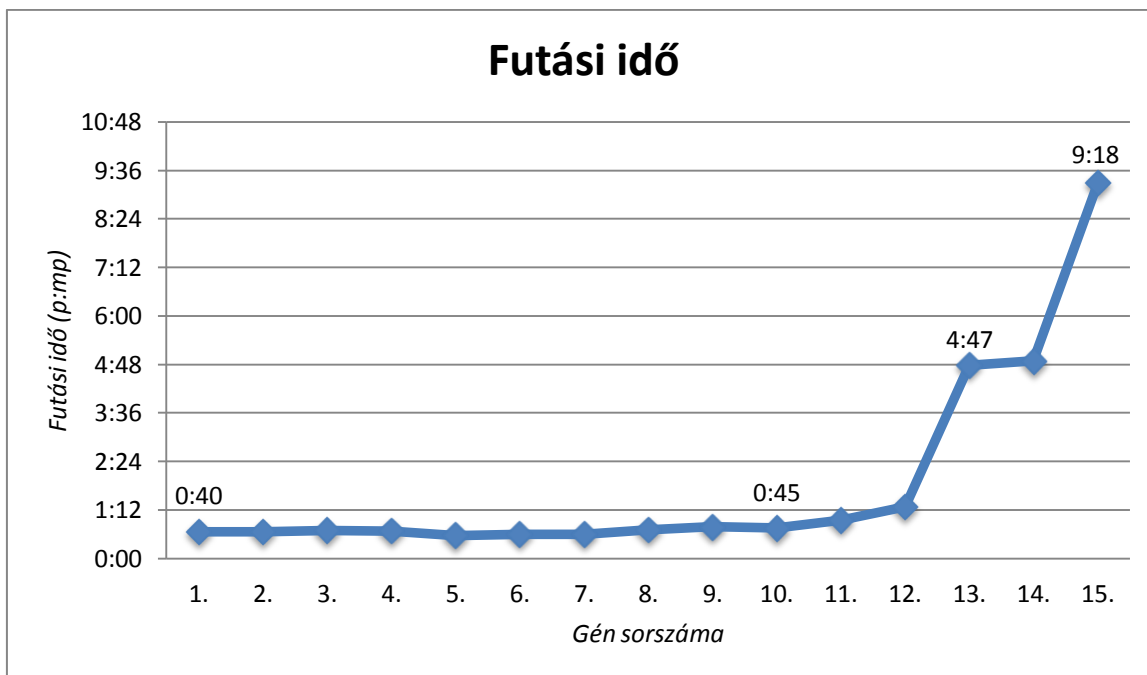
A futási idő nagyban függ attól, hogy milyen gének interakcióira vagyunk kíváncsiak, hiszen egy génhez 4-5 sortól kezdődően akár több száz sornyi adat is szerepelhet a kimeneti fájlunkban. A tesztelés alatt csak az MR job-ok futási idejét mértem, emellett az output fájlra

vonatkozólag gyűjtöttem össze néhány fontos adatot. Az alábbi táblázat 1-től kezdődően, egészen 15 db génig tartalmazza az eredményeket:

Gének száma	Kimeneti sorok száma	Kimeneti fájl mérete	Futási idő
1	3	604 byte	0:40
2	12	2.5 kB	0:40
3	170	32.4 kB	0:42
4	384	75.7 kB	0:41
5	724	145.7 kB	0:34
6	1552	319.7 kB	0:36
7	1576	326.3 kB	0:36
8	1976	417.9 kB	0:43
9	3608	773.0 kB	0:47
10	4080	884.9 kB	0:45
11	9056	2.4 MB	0:57
12	11840	3.4 MB	1:17
13	78912	40.8 MB	4:47
14	85440	42.3 MB	4:53
15	169856	84.9 MB	9:18

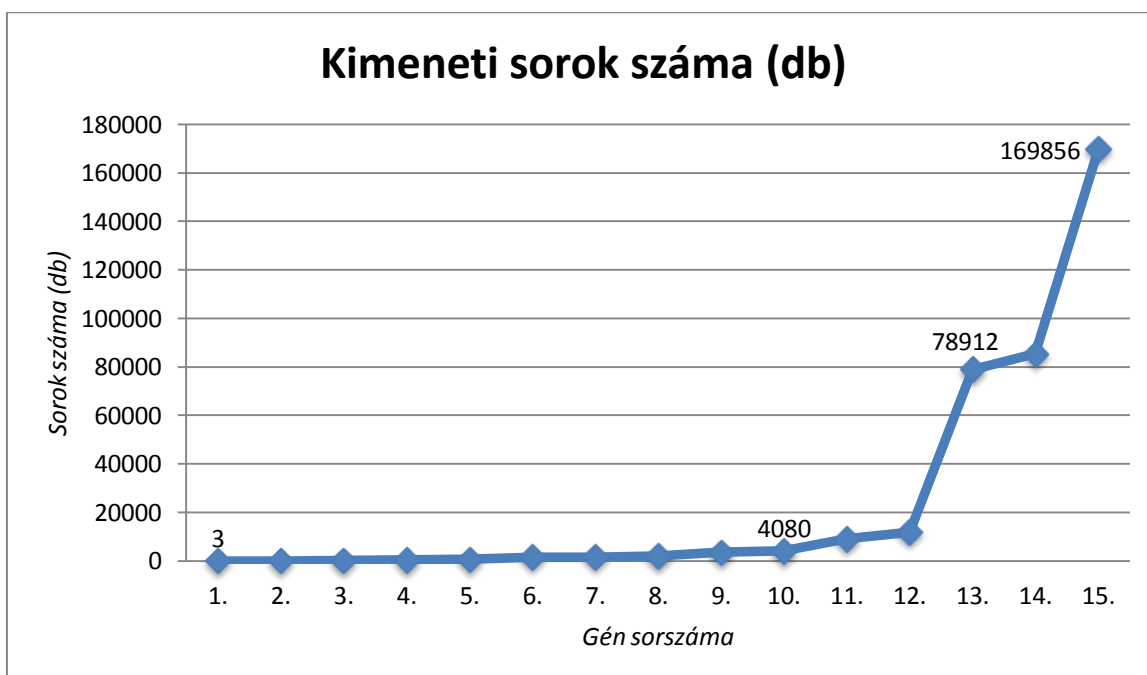
*3. táblázat - Mérési eredmények
Észrevehető, hogy mind a futási idő és a kimeneti fájl mérete
(sorok számával egyetemben) a 13. géntől kezdődően növekedett meg nagymértékben*

A futási idő egészen 12. génig nem számottevő (csupán 1 perc körüli), pedig már ennyi gén esetén is 4095 lekérdezésről beszélünk. 15 gén esetén a program majdnem 10 percet futott, számomra ez volt az a határ ahonnan már legalább egy két node-os klaszter kiépítése szükséges lett volna. Ugyanis két erőforrás használatakor a futási idő megfelelőzne, ráadásul ehhez a kódot és a futtatási paramétereket sem kellene módosítanunk. Sajnos egy ilyen rendszeren való tesztelésre nem volt lehetőségem.

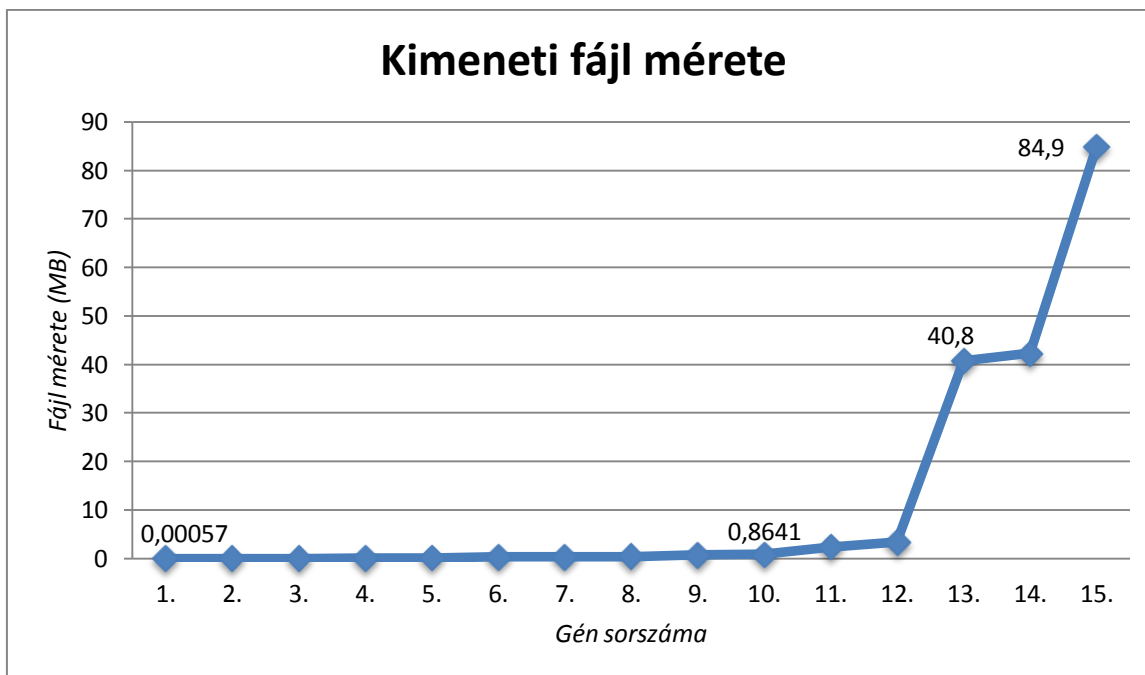


1. diagram - A futási idő egészen a 12. génig nem számottevő, viszont a Cyp6a19 géntől kezdődően több percig futott a program.

Az output file méretét és sorainak számát elsősorban a REST service-től fogadott válaszok száma határozza meg, melyet a 2. diagram reprezentál. A futási időhöz hasonlóan a kimeneti fájl sorainak száma és mérete a 13. géntől kezdődően növekedett meg drasztikusan.



2. diagram - A kimeneti sorok száma a 13. géntől kezdve majdnem hétszeresére nőtt



3. diagram - A kimeneti fájl mérete, a sorokhoz hasonló ütemben változott. A 15. génnél éppen nem éri el a 85MB-ot. Ekkora adatmennyiség még nem tekinthető nagy adathalmaznak, viszont a 10 perces futási idő sajnos egy ilyen rendszeren már nem tekinthető hatékonynak.

Összességében megállapítható, hogy a szoftver egy single-node-os rendszeren is tud hatékony működést produkálni. Egyedül a futási idő volt az, ami nagymértékben befolyásolta ezt a hatékonyságot, hiszen a program 15 gén esetén már majdnem 10 percig futott. Azonban a kimeneti fájl mérete (és a sorok száma) egyáltalán nem haladta meg egy hétköznapi számítógép erőforrásait. Esetleg ha a vizsgált génmennyiségtől többet szeretnénk lekérdezni, a futási idő csökkentésének érdekében a programot érdemes lehet egy néhány node-os klaszter-rendszeren keresztül futtatni, több száz és több ezer gén esetén pedig egy nagyobb Hadoop klaszter kiépítése lehet a legcélszerűbb döntés.

4. Összefoglalás

A funkcionális genomika az egyes gének biológiai folyamatokban betöltött szerepét vizsgálja. Az ezen a területen végzett kutatások elengedhetetlen feltétele a kísérletek alapos megtervezése, azonban az egy kísérletben potenciálisan vizsgált gének és a köztük fennálló ismert kapcsolatok száma rendkívül magas, és a kísérleti biológusok számára nehézséget jelent ezen információk beszerzése és rendszerezése. Erre a problémára igyekszik megoldást kínálni a dolgozatban bemutatott szoftver, amely egy felhasználó által megadott génlistára összegyűjti és rendszerezi a gének közt fennálló kapcsolatokat.

A szoftver az Apache Hadoop nevű nyílt forráskódú keretrendszeren alapszik, amellyel lehetővé vált az adat-intenzív elosztott alkalmazásoknak a nagyméretű és egyben alacsony költségű grid infrastruktúrával történő támogatása. Az architektúra biztosít egy elosztott fájlrendszert - a HDFS-t - és emellett implementálja a MapReduce paradigmát is, amit a funkcionális programozás map és reduce funkciói inspiráltak. A gének kinyeréséhez egy BioGRID nevű adatbázis REST alapú webszolgáltatását használtam, amely a fehérje-fehérje- és géninterakcióknak az egyik legnagyobb "gyűjtőhelye".

A Hadoop (és így az alkalmazásom) egyik legnagyobb előnye a skálázhatóság, azaz a programot elosztott környezetben futtatva a bemenet méretének függvényében exponenciálisan növekvő lehetséges kombinációkat, lineárisan skálázva dolgozhatjuk fel. A használatához csupán egy egyszerű szöveges fájlt kell megadnunk, majd néhány egyszerű lépés múlva már a saját lokális fájlrendszerünkön találjuk azt a kimeneti fájlt, amely a számunkra fontos gének (egyes, páros, hármas, stb.) interakcióit tartalmazza. Továbbá a programhoz implementálva lett egy egyszerű grafikai felület is, amely lehetővé teszi az alkalmazás paraméterezését is, annak mélyreható ismerete nélkül. A végső output formátuma pedig olyan CSV fájl lesz, melyet akár azonnal importálhatunk bármilyen táblázatkezelő programba vagy akár egy egyszerű notepad-el is megnyithatjuk.

5. Irodalomjegyzék

- [1] *Az Apache Hadoop hivatalos weboldala*
<http://hadoop.apache.org/>
[Letöltve: 2013. augusztus 17.]
- [2] deRoss, D., Zikopoulos, P., C., Brown, B., Coss, R., Melnyk, R., B., (2014), *Hadoop for Dummies*, John Wiley & Sons Inc., pp 115-116
- [3] *LSI technologies in data storage used for Higgs boson searches at CERN* (2002)
<https://www.lsi.com/downloads/Public/Host%20Bus%20Adapters/LSI-CS-CERN.pdf>
[Letöltve: 2014. október 8.]
- [4] *Obama Administration Unveils “Big Data” Initiative: Announces \$200 million in new R&D investments* (2012)
http://www.whitehouse.gov/sites/default/files/microsites/ostp/big_data_press_release_final_2.pdf
[Letöltve: 2014. október 8.]
- [5] *RFC 4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files* (2005)
<http://tools.ietf.org/html/rfc4180>
[Letöltve: 2014. október 8.]
- [6] White, T., (2009), *Hadoop: The Definitive Guide*, O'Reilly media, pp 9-12
- [7] *Yahoo! Hadoop Tutorial*
<https://developer.yahoo.com/hadoop/tutorial/>
[Letöltve: 2013. augusztus 17.]
- [8] *Wikipedia - Apache Hadoop*
http://en.wikipedia.org/wiki/Apache_Hadoop
[Letöltve: 2013. október 19.]
- [9] *Wikipedia - REST*
<http://hu.wikipedia.org/wiki/REST>
[Letöltve: 2014. október 8.]

6. Ábrajegyzék

A dolgozatban található ábrák nagy része külföldi dokumentációkban található képek fordításai alapján készültek. Az alábbi jegyzék a dolgozatban szereplő ábrák mellett az eredeti képek linkjeit (is) tartalmazza.

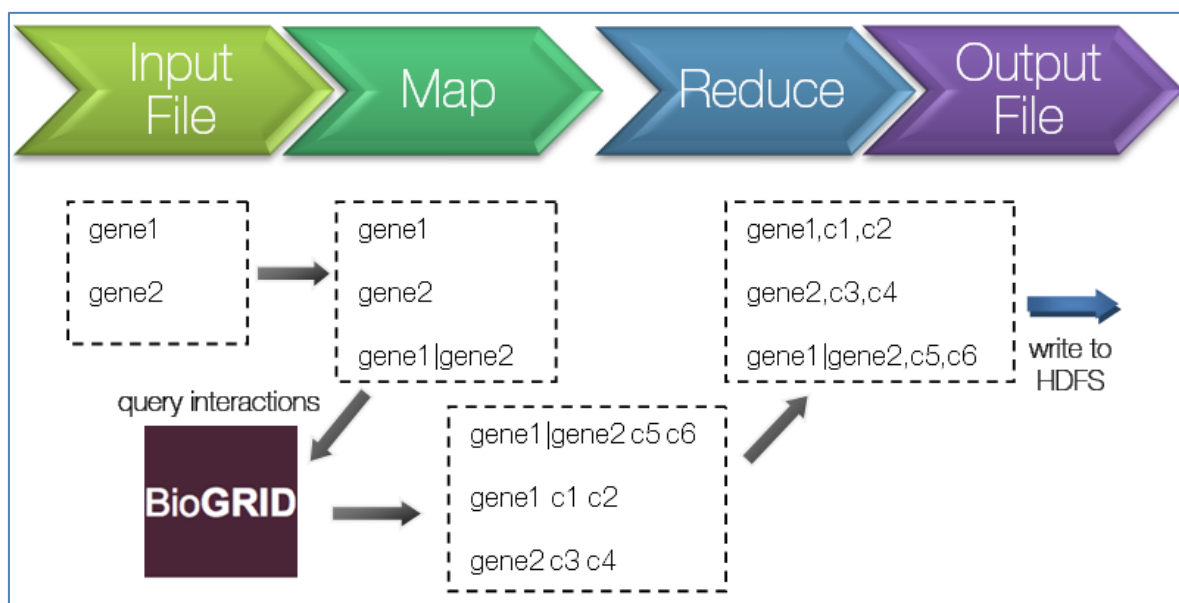
1. ábra *Fájlblokkok tárolása a DataNode-okon*
https://farm3.static.flickr.com/2050/3529146393_5c2e2c8065_o.png
[Letöltve: 2014. október 8.]
2. ábra *A "word count" program működése*
<http://3.bp.blogspot.com/-WWpJagRwDzc/Ualo2IZhUpI/AAAAAAAAACQ/95lFQf5dNv8/s640/HadoopEcosystem.png>
[Letöltve: 2014. október 8.]
3. ábra *A Job Tracker és a Task Tracker kapcsolata*
<http://cfile25.uf.tistory.com/image/1762633B4F571A051B3CE3>
[Letöltve: 2014. október 8.]
4. ábra *Hadoop Ecosystem*
<http://yourstory.com/wp-content/uploads/2012/04/Hadoop-Ecosystem.png>
[Letöltve: 2014. október 9.]
5. ábra *Információk egy MR job-ról*
6. ábra *BioGRID és a Hadoop logója*
<http://www.virtuallyimmune.org/wp-content/uploads/2014/07/BioGRID.png>
<https://www1.clouditcanada.com/wp-content/uploads/2013/12/sticker375x360.u4.png>
[Letöltve: 2014. október 9.]
7. ábra *A program működése*
8. ábra *A MapReduce folyamata*
9. ábra *Könyvtárszerkezet*
10. ábra *A szoftver kezdőképernyője*
11. ábra *Az input fájl és az output könyvtár kiválasztását segítő dialógusablak*
12. ábra *Hibaüzenet ablak*

13. ábra *A MR job indítása előtt egy végső "Final check" ablak jelenik meg, ahol ellenőrizni tudjuk a beállításainkat.*
14. ábra *Az MR job futásának végét jelző ablak*
15. ábra *CDC27, APC1 és APC2 génhez szükséges link felépítése*
<http://wiki.thebiogrid.org/doku.php/biogridrest>
[Letöltve: 2014. október 9.]
16. ábra *REST architektúra*
<http://www.scriptol.com/programming/images/rest-architecture.jpg>
[Letöltve: 2014. október 9.]
17. ábra *Importált CSV fájl a LibreOffice Calc programban*
18. ábra *A szoftver MapReduce részének működése*

7. Eredmények

Munkám során egy **olyan szoftvert alkottam, amely a funkcionális genomikával foglalkozó kísérleti biológusok munkáját igyekszik segíteni**. Az ezen a területen végzett kutatások elengedhetetlen feltétele - a kísérletek alapos megtervezése mellett -, az egy kísérletben potenciálisan vizsgált gének és a köztük fennálló ismert kapcsolatoknak a meghatározása. Az általam írt alkalmazás erre kíván megoldást nyújtani, amely **egy felhasználó által megadott génlistára összegyűjti és rendszerezi a gének közt fennálló kapcsolatokat és olyan formátumba menti el**, amely könnyen használhatóvá teszi, például egy táblázatkezelő program számára. A megvalósításhoz az **Apache Hadoop keretrendszert** használtam, amely jelenleg az egyik legmodernebb eszköze az adatintenzív alkalmazások támogatásának. Alapját egy elosztott fájlrendszer (*HDFS*) és a *MapReduce* paradigma határozza meg.

Elsőként a Hadoop keretrendszert telepítettem egy virtuális *Ubuntu Linux* operációs rendszerre, melynek konfigurálását magam végeztem el. Ezután a **Hadoop Streaming utility segítségével egy olyan Python alapú MapReduce programot fejlesztettem**, amely gének interakcióit képes lekérdezni a *BioGRID REST* alapú webservice-en keresztül. A *mapper.py* részben **olyan map függvényt definiáltam, amely az egyes URL-ek és a REST hívások elosztott kezelését teszi lehetővé**. A rendezés alapját a részt vevő gének alfabetikus sorrendje határozza meg. A *reducer.py* a **beérkező információ rendszerezését és megfelelő formátumúvá alakítását végző reducer függvényt tartalmazza**. A könnyű használhatóságot szem előtt tartva egy **felhasználói felület is implementálásra került** a Python *easyGUI* moduljának segítségével. A GUI segítségével bárki könnyedén tudja a szoftvert használni, csupán néhány kattintás segítségével. A kimeneti fájl formátumát CSV-nek választottam, ezért egy rövid *Java* kód is belekerült a többi kód mellé, amely minden kimeneti fájl kiterjesztését .csv-re változtatja. Ezek után **tesztelést végeztem arra vonatkozóan, hogy meddig érdemes a szoftvert egy single-node-os rendszeren futtatni, azaz milyen génmennyiségnél ajánlott egy "többnode"-os klaszter kialakítása**. Többek között a futási időt és a kimeneti fájl méretét vizsgáltam. Az elkészült alkalmazásom jelenlegi változata szabadon elérhető az alábbi github repository-ban: (<https://github.com/benauer/BioGrid-Hadoop>).



18. ábra - A szoftver MapReduce részének működése

Főbb eredmények összesítve:

- Single-node-os Hadoop-klaszter létrehozása
- MapReduce program fejlesztése:
 - *mapper.py*: az interakciók lekérdezése a BioGRID REST webszolgtatás segítségével
 - *reducer.py*: a lekérdezett adatok rendezése és a CSV formátumának való megfeleltetése (RFC 4180)
 - *start.py*: grafikus felület implementálása a könnyű használhatóságért
 - *CustomCSVFormat.java*: rövid kód a kimeneti fájl kiterjesztésének eléréséért
- Publikálás: <https://github.com/benauer/BioGrid-Hadoop>
- Tesztelés: a hatékony működés vizsgálata egy single-node-os rendszeren